第五章 树与二叉树

一. 树的基本概念

1. 树的定义：树是一种数据结构，它是由n个有限节点组成一个具有层次关系的集合

2. 树的特点：① 每个节点有零个或多个子节点

② 没有父节点的节点称为根节点

③ 每一个非根节点有且只有一个父节点

④ 除了根节点外，每个子节点可以分为多个不相交的子树.

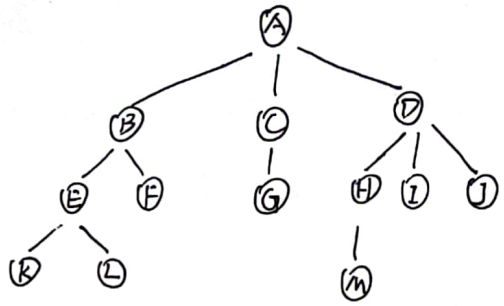
3. 基本术语：

(1) 结点的度：树中一个结点的孩子个数.

树的度：树中结点的最大度数.

(2) 分支结点：非终端结点，度大于0.

叶结点：终端结点，度为0



第一层

高度4

第四层.

(3) 结点的深度：结点所在的层次.

树的高度(深度)：树中结点的最大层数.

结点的高度：以该结点为根的子树的高度.

(4) 森林：n个或多个不相交的树组成

(5) 有序树：树中结点的各子树从左到右有次序不能互换.

(6) 无序树：树中结点可以互换位置.

(7) 路径长度：路径上所经过的边的个数 (两结点之间的结点序列)

4. 树的性质

(1) 树的结点数=所有结点的度数之和+1.

(2) 度为m的树中第i层上最多有m^{i-1}个结点.

(3) 已知度m和高度h
- 求树的最少结点数：让1~h-1层结点数都为1，最后一层为m，h+m-1个结点.
- 求树的最多结点数：满m叉树，$1+m+m^2+\cdots+m^{h-1}=\frac{m^h-1}{m-1}$个结点.

(4) 已知度m和结点个数n
- 最小高度 $h=\lceil \log_m(n(m-1)+1)\rceil$.
- 最大高度 $h=n-m+1$

二. 二叉树

1. 定义：二叉树是每个结点至多只有两棵子树的树结构。（不存在度>2的结点）
① 有序树，左、右子树次序不能指
② 可以是空集

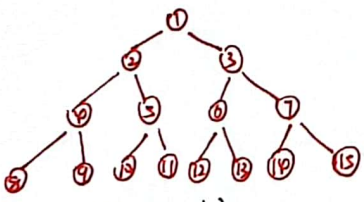
△五种基本形态：

△二叉树与度为2的有序树的区别：
① 二叉树可空，~至少有3个结点。
② 这二叉树结点次序是确定的，
~的孩子的左右次序是相对于另一个孩子而言。

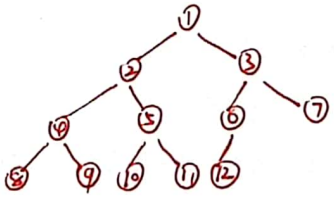
2. 性质：
① 二叉树第i层上的结点数目最多为 2^{i-1} 个。
② 深度为k的二叉树最多有 2^k-1 个结点。
③ 包含n个结点的二叉树的高度至少为 $\lceil \log_2(n+1) \rceil$ 或 $\lfloor \log_2 n \rfloor + 1$
④ 树的结点数 = 所有结点的度数之和 + 1
⑤ 任意一颗二叉树，终端结点个数 n_0，度为2结点个数 n_2，则 $n_0 = n_2 + 1$。

3. 特殊之二叉树
(1) 满二叉树：高度h，且有 2^h-1 个结点。
① 除叶结点，每个结点度数均为2。
② 层序编号，根为1，自上而下，对于i：
双亲 $\lfloor i/2 \rfloor$
左孩子 $2i$
右孩子 $2i+1$

(2) 完全二叉树：① 若 $i \le \lfloor n/2 \rfloor$，则结点i为分支结点，否则为叶结点。
② 叶结点只能出现在最下层和次下层，最下层的叶结点依次排列在左边。
③ 度为1的结点数：0或1。
④ 满二叉树 ⇒ 完全二叉树

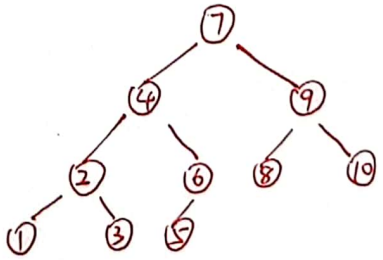


（满二叉树）



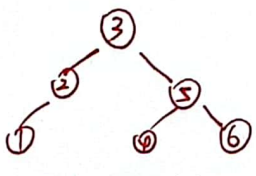
（完全二叉树）

(3) 二叉查找树 [二叉排序树] [二叉搜索树]：
① 左子树结点关键字小于根结点。
② 右子树结点关键字大于根结点。
③ 没有键值相等的结点。



（二叉查找树）

(4) 平衡二叉树：
树上任一结点的左子树和右子树的深度之差不超过1
绝对值



（平衡二叉树）

(5) 正则二叉树：树中每个分支结点都有2个孩子，即树中只有度为0或2的结点。

4. 二叉树存储结构

(1) 顺序存储：一般用数组存二叉树的结点，自上而下，自左至右 → 一维数组

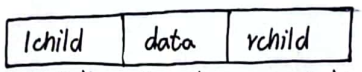
① 节点x存储在下标为i的位置 { 左：2i / 右：2i+1 / 父：⌊i/2⌋ }

只要知道根结点的存储位置，就可把整棵树串起来。

②空间利用率不高。

③适用于 { 完全二叉树 / 满二叉树 }

(2)链式存储：



左指针域 数据域 右指针域

至少包含3个域

只要知道根结点，就可以通过左右子结点的指针把整棵二叉树串起来。

```
typedef  struct  BiTNode {
    ElemType  data;
    struct  BiTNode  *lchild, *rchild;
} BiTNode, *BiTree;
```

n个结点的二叉链表中，含有n+1个空链域。

适用于二叉树

5. 二叉树的遍历

(1) 先序遍历 根左右

①递归：

```
void  PreOrder (BiTree  T) {
    if (T != NULL) {
        visit (T);
        PreOrder (T->lchild);
        PreOrder (T->rchild);
    }
}
```

②非递归：

```
void  PreOrder2 (BiTree  T) {
    InitStack(S);  BiTree  p=T;  //初始栈S
    while (p || !IsEmpty (S)) {  //栈非空或p不空
        if(p) {                  //一路向左
            visit(p);  Push(S,p);  //访问当前结点,入栈.
            p=p->lchild;           //遍历左子树
        }
        else {
            Pop(S,p);              //出栈,栈顶元素
            p=p->rchild;           //遍历右子树.
        }
    }
```

(2)中序遍历 左根右

①递归：

```
void  InOrder (BiTree T) {
    if (T != NULL) {
        InOrder (T->lchild);
        visit (T);
        InOrder (T->rchild);
    }
}
```

②非递归：

```
void  InOrder2 (BiTree  T) {
    InitStack(S);  BiTree  p=T;  //初始栈S
    while (p || !IsEmpty (S)) {  //栈不空
        if(p) {                  //一路向左
            Push(S,p);           //当前结点入栈
            p=p->lchild;         //遍历左子树
        }
        else {                   //出栈,并转向栈结点的右子树
            Pop(S,p); visit(p);  //栈顶元素出栈,并访问
            p=p->rchild;         //遍历右子树
        }
    }
}
```

(3)后序遍历 左右根

①递归:

```
void PostOrder(BiTree T){
    if(T!=NULL){
        PostOrder(T->lchid);
        PostOrder(T->rchild);
        visit(T);
    }
}
```

②非递归:

```
void PostOrder2(BiTree T){
    InitStack(S);
    BiTNode *p=T;
    BiTNode *r=NULL;
    while(p||!IsEmpty(S)){
        if(p){              //走到最左边
            push(S,p);      //左孩子依次入栈
            p=p->lchild;    //直到左孩子空
        }
        else{               //向右
            GetTop(S,p);    //读栈顶结点
            if(p->rchild&&p->rchild!=r)
                p=p->rchlid;//右子树存在且未被访问,转向右.
            else{           //否则弹出结点并访问
                Pop(S,p);
                visit(p->data);
                r=p;        //记录最近访问过的结点
                p=NULL;     //每次出栈访问完一个结点就
            }               //相当于遍历完以该结点为根的
        }                   //子树,需将p置NULL
    }
```

后序遍历可应用于:求根到某结点的路径,
求两个结点的最近公共祖先等。

(4)层次遍历 从上至下,从左至右访问所有结点。

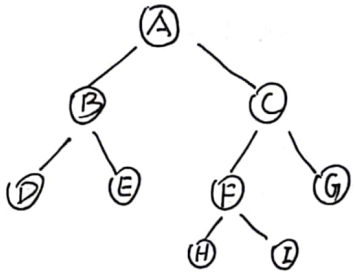
(基于队列实现)

①根结点入队。

②队列非空,队头结点出队,访问该结点;
左孩子入队,右孩子入队。

③重复②,直至队列空.

```
void LevelOrder(BiTree T){
    InitQueue(Q);
    BiTree p;
    EnQueue(Q,T);
    while(!IsEmpty(Q)){
        DeQueue(Q,p);
        visit(p);
        if(p->lchild!=NULL)
            EnQueue(Q,p->lchild);
        if(p->rchild!=NULL)
            EnQueue(Q,p->rchild);
    }
}
```



前序: ABDECFHIG

中序: DBEAHFICG

后序: DEBHIFGCA

层次: ABCDEFGHI

递归形式 时间复杂度 O(n) 每个结点都访问一次且仅一次.
空间复杂度 O(n) 递归工作栈的栈深为树的深度

结论: ①不能唯一确定一颗二叉树的是:先序+后序

②前序序列和中序序列的关系相当于:以前序序列为入栈次序,以中序序列为出栈序列.

③前序序列和后序序列相反:二叉树高度=结点树.

④后序遍历可以找到m到n的直接路径(m是n的祖先).

6. 线索二叉树

(1) 基本概念：

① 对一颗二叉树中所有结点的空指针域按照某种遍历方式加线索的过程叫作线索化。

② 线索二叉树是一种物理结构。

③ 引入线索二叉树的目的：加快查找结点的前驱或和后继的速度。

④ n个结点的线索二叉树上含有线索数量为 n+1 个。

⑤ 利用二叉树的 n+1 个空指针来存放结点的前驱和后继信息。

(2) 结点结构：

lchild	ltag	data	rtag	rchild

$$ltag = \begin{cases} 0, & lchild \text{ 指向结点的左孩子} \\ 1, & lchild \text{ 指向结点的直接前驱} \end{cases} \qquad rtag = \begin{cases} 0, & rchild \text{ 指向结点的右孩子} \\ 1, & rchild \text{ 指向结点的直接后继} \end{cases}$$

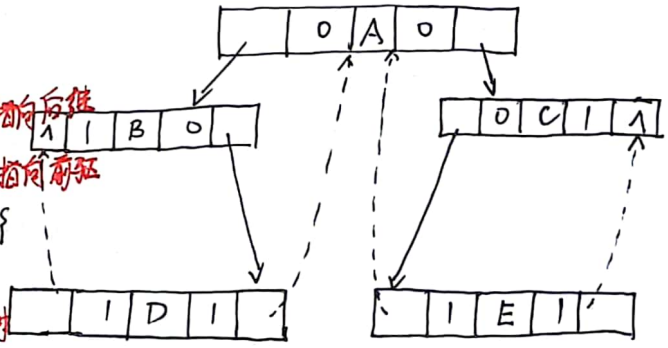
```
typedef  struct  ThreadNode{
    ElemType  data;
    struct  ThreadNode  *lchild, *rchild;
    int  ltag, rtag;
}ThreadNode, *ThreadTree;
```

(3) 中序线索二叉树

① 中序遍历

② 结点右子结点为空的指针域指向后继

③ 结点左子结点为空的指针域指向前驱



① 线索化递归：

```
void  InThread(ThreadTree &p, ThreadTree &pre){
    if(p!=NULL){
        InThread(p->lchild, pre);  //递归，线索化左子树
        if(p->lchild==NULL){  //左子树为空
            p->lchild=pre;  //建立前驱线索
            p->ltag=1;
        }
        if(pre!=NULL && pre->rchild==NULL){  //前驱结点非空且右子树空
            pre->rchild=p;  //建立前驱结点的后继线索
            pre->rtag=1;
        }
        pre=p;  //标记当前结点成为刚刚访问过的结点
        InThread(p->rchild, pre);  //递归，线索化右子树
    }
}
```

△ 前序和后序线索化只需要把遍历方法改成前序和后序即可

② 主过程：

```
void  CreateInThread(ThreadTree T){
    ThreadTree  pre=NULL;
    if(T!=NULL){
        InThread(T, pre);  //线索化
        pre->rchild=NULL;  //处理遍历的最后一个结点.
    }
}
```

(4) 遍历(中序线索二叉树)

① 求中序线索二叉树的中序序列下的第一个结点:

```
ThreadNode *FirstNode(ThreadNode *p){
    while(p->ltag==0) p=p->lchild;   //最左下结点(不一定是叶结点)
    return p;
}
```

② 求中序线索二叉树中结点p在中序序列下的后继:

```
ThreadNode *NextNode(ThreadNode *p){
    if(p->rtag==0) return FirstNode(p->rchild); //右子树中最左下结点
    else return p->rchild;  //rtag==1,直接返回后继
}
```

③ 利用上面两个算法,可写出不含头结点的中序线索二叉树的中序遍历的算法:

```
void Inorder(ThreadNode *T){
    for(ThreadNode *p=FirstNode(T); p!=NULL; p=NextNode(p))
        visit(p);
}
```

(5) 结论:

①在中序线索树中,若某结点有右孩子,则其后继结点是它的右子树的最左下结点。

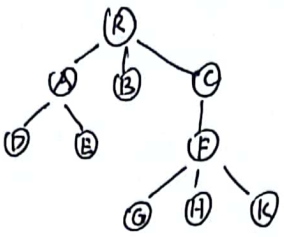
②在中序线索树中,若某结点有左孩子,则其前驱结点是它的左子树的最右下结点。

③ 先序线索树的后继
- a.有左孩子,则左孩子是其后继;
- b.有右孩子(无左孩子),右孩子是其后继;
- c.叶结点,右链域直接指示了结点后继。

④后序线索树的后继(采用带标志域的三叉链表作为存储结构)
- a.结点x是根,后继为空;
- b.结点x是其双亲的右孩子,或是其双亲的左孩子且其双亲没有右子树,后继为双亲;
- c.结点x是其双亲的左孩子,且其双亲有右子树,则其后继为双亲的右子树上按后序遍历列出的第一个结点.

二、树、森林

1. 树的存储结构



	data	parent
0	R	-1
1	A	0
2	B	0
3	C	0
4	D	1
5	E	1
6	F	3
7	G	6
8	H	6
9	K	6

(1) 双亲表示法：

① 采用一组连续空间来存储每个结点

② 在每个结点中设置一个伪指针

③ 伪指针指示其双亲结点在数组中的位置.

④ 利用了每个结点（根结点除外）只有唯一双亲的性质，可以很快地得到每个结点的双亲结点，但求结点的孩子时则需遍历整个结构.

```
#define MAX_TREE_SIZE 100
typedef struct {            //树的结点定义
    ElemType data;          //数据元素
    int parent;             //双亲位置域
}PTNode;
typedef struct {            //树的类型定义
    PTNode nodes[MAX_TREE_SIZE]; //双亲表示
    int n;                  //结点数
}PTree;
```

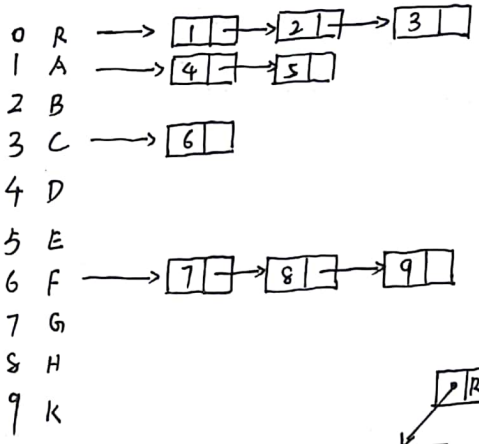
(2) 孩子表示法：

① 将每个结点的孩子结点视为一个线性表

② 单链表作为存储结构

③ n个结点就有n个孩子链表.

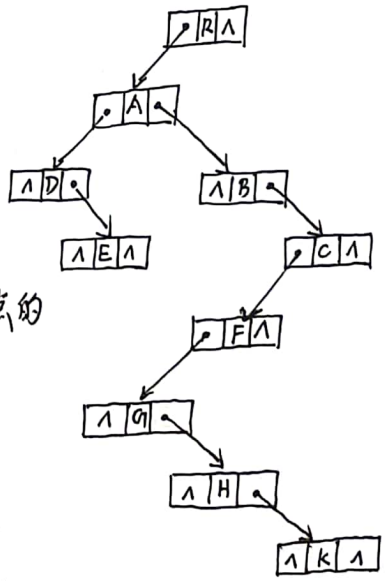
④ 便于寻找孩子，寻找双亲则需遍历n个结点中孩子链表指针域所指向的n个孩子链表.



(3) 孩子兄弟表示法（二叉树表示法）：

① 以二叉链表作为树的存储结构

② 结点内容包含：结点值；指向结点第一个孩子结点的指针；指向结点下一个兄弟结点的指针（沿此域可以找到结点的所有兄弟结点）




```
typedef struct CSNode{
    ElemType data;
    struct CSNode *firstchild, *nextSibling; //第一个孩子和右兄弟指针
}CSNode, *CSTree;
```

③ 可以方便地实现树转换为二叉树的操作，易于查找结点的孩子.

④ 缺点是从当前结点查找其双亲比较麻烦，可为每结点增设一个 parent 域指向父结点。

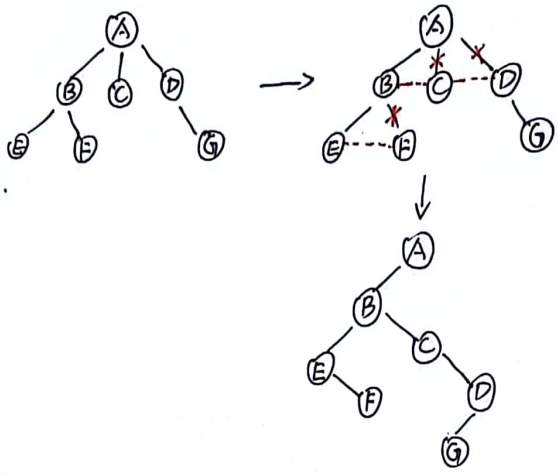
2. 树、森林与二叉树的转换

(1) 树转换为二叉树：

① 在兄弟结点之间加一连线.

② 对每个结点，只保留它与第一个孩子的连线.

③ 以树根为轴心，顺时针旋转45度.

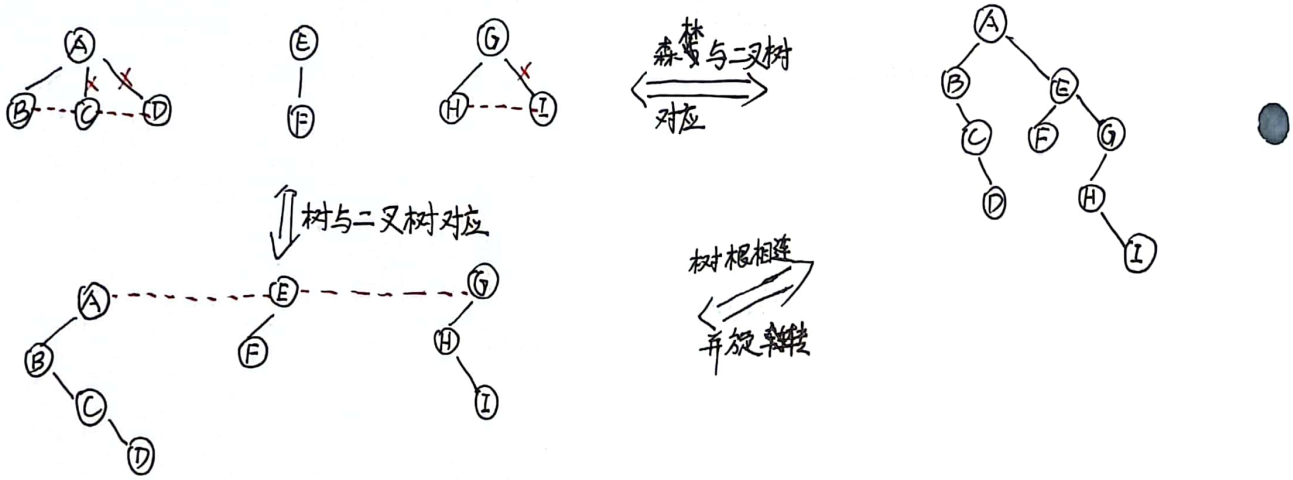


(2) 森林转换为二叉树：

① 将森林中的每棵树转换成相应的二叉树.

② 每棵树的根也可以视为兄弟关系，在每棵树的根之间加一根连线.

③ 以第一根树的根为轴心旋转45°.



(3) 二叉树转换为森林：

二叉树转换为树或森林是惟一的.

① 根及其左子树为第一棵树的二叉树形式，将根的右链断开.

② 根的右子树的右链断开，即为第二棵树的二叉树形式.

③ 直到只剩一棵没有右子树的二叉树为止，最后树成森林.

3. 树和森林的遍历

树	森林	二叉树
先根遍历	先序遍历	先序遍历
后根遍历	中序遍历	中序遍历

(1) 树：

① 先根遍历 ABEFCDG：先访问根结点；再依次遍历根结点的每棵子树.

② 后根遍历 EFBCGDA：先依次遍历根结点的每棵子树；再访问根结点.

(2) 森林：

① 先序遍历：访问森林中第一棵树的结点；先序遍历第一棵树中根结点的子树森林；先序遍历除去第一棵树之后剩余的树构成的森林。 ABCDEFGHI

② 中序遍历：中序遍历森林中第一棵树的根结点的子树森林；访问第一棵树的根结点；中序遍历除去第一棵树之后剩余的树构成的森林 BCDAFEHIG

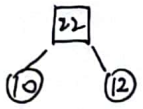
四. 哈夫曼树

1. 定义：树的带权路径长度(WPL)最小的二叉树。 $WPL=\sum_{i}^{n} wili$

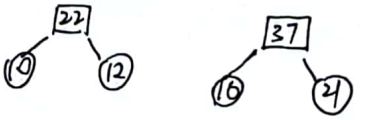
wi是第i个叶结点所带的权值；
li是该叶结点到根结点的路径长度。

2. 构造：每次把队列中值最小的两个合并，合并的值放入队列中再继续比较。

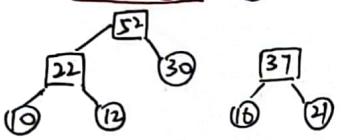
(1) 10 12 16 21 30



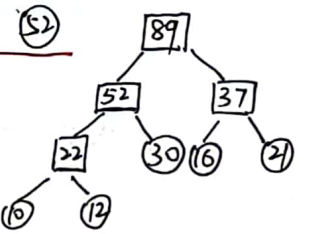
(2) 16 21 22 30



(3) 22 30 37



(4) 37 52



$WPL=(10+12)\times3+(30+16+21)\times2$
$=200$

3. 特点：
① 没有度为1的结点；
② n个叶结点的哈夫曼树共有2n-1个结点；
③ 任意非叶结点的左右子树交换后仍是哈夫曼树；
④ 对同一组权值，可能存在不同构的多棵哈夫曼树；
⑤ 哈夫曼树不一定是完全二叉树。

4. 哈夫曼编码：

(1) 相关概念：

① 前缀编码：没有一个编码是另一个编码的前缀
② 固定长度编码：每个字符用相等长度的二进制表示。
③ 可变长度编码：允许对不同字符用不等长的二进制位表示。
- 对频率高的字符赋以短编码
- 字符的平均编码长度减短，压缩数据。

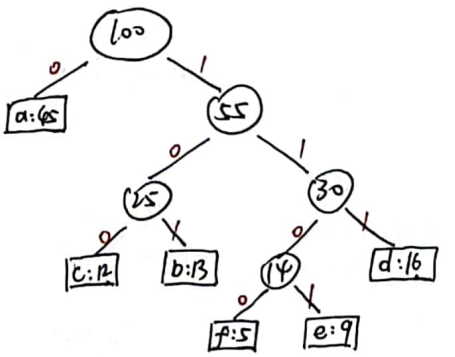
④ 哈夫曼编码：将每个字符当作一个独立的结点，其权值为它出现的频度，构造出相应的哈夫曼树；然后，将从根到叶结点的路径上分支标记的字符串作为该字符的编码。

$WPL=1\times45+3\times(13+12+16)+4\times(5+9)$
$=224$

可视为最终编码得到二进制编码的长度，共224位。

若采用3位固定长度编码，300位。

各字符编码为：
a：0
b：101
c：100
d：111
e：1101
f：1100



五. 并查集

1. 概念：一种简单的集合表示，支持3种操作：
   - Initial(S)
   - Union(S, Root1, Root2)
   - Find(S, x)

2. 存储结构：通常用树的双亲表示作为并查集的存储结构。
   ① 每个子集合以一棵树表示。
   ② 所有表示子集合的树，构成表示全集合的森林，存放在双亲表示数组内。
   ③ 数组元素的下标代表元素名，
   ④ 根结点的下标代表子集合名。
   ⑤ 根结点的双亲域为负数（可设置为该子集合元素数量的相反数）。

```
#define SIZE 100
```

3. 基本实现：`int UFSets[SIZE];` //集合元素数组（双亲指针数组）

(1) Initial(S)：将集合S中的每个元素都初始化为只有一个单元素的子集合。

```
void Initial(int S[]){
    for(int i=0; i<SIZE; i++)
        S[i] = -1;
}
```

S ⓪ ① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨

(a) 全集合S初始化时形成一个森林

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

(b) 初始化时形成的（森林）双亲表示

(2) Union(S, Root1, Root2)：时间 ~ O(1)
① 把集合S中的子集合Root2并入Root1；
② 要求Root1和Root2互不相交，否则不执行合并。

```
void Union(int S[], Root1, Root2){
    if(Root1 == Root2) return;
    S[Root2] = Root1;   //将根Root2直接到根Root1下面
}
```

S_1 ⓪ ⑥ ⑦ ⑧　S_2 ① ④ ⑨　S_3 ② ③ ⑤

(c) 集合的树形表示

0	1	2	3	4	5	6	7	8	9
-4	-3	-3	2	1	2	0	0	0	1

(d) 集合S_1, S_2和S_3的（森林）双亲表示

时间 ~ O(d)　d为树深

(3) Find(S, x)：查找集合S中单元素x所在的子集合，并返回该子集合的根结点。

```
int Find(int S[], int x){
    while(S[x] >= 0)      //循环寻找x的根
        x = S[x];
    return x;             //根的S[]小于0
}
```

4. 优化：

(1) Union：把小树合并到大树，深度不超过$\lfloor \log_2 n \rfloor + 1$。

```
void Union(int S[], int Root1, int Root2){
    if(Root1 == Root2) return
    if(S[Root2] > S[Root1]){   //Root2结点数更少
        S[Root1] += S[Root2];  //累加集合Root2的总结点数
        S[Root2] = Root1;      //小树并大树
    }
    else{
        S[Root2] += S[Root1];
        S[Root1] = Root2;
    }
}
```

集合树深不超过$O(\alpha(n))$，其中$\alpha(n)$增长极其缓慢，通常$\alpha(n) \le 4$。

(2) Find：当所查元素x不在树的第二层时，在算中增加一个"压缩路径"的功能，即将从根到元素路径上的所有元素都变成根的孩子。

```
int Find(int S[], int x){
    int root = x;
    while(S[root] >= 0)  root = S[root];   //循环找到根
    while(x != root){                      //压缩路径
        int t = S[x];   //t指向x的父结点
        S[x] = root;    //x直接挂到根结点下面
        x = t;
    }
    return root;        //返回根结点编号
}
```